AIAA-98-4532

MANAGING SHARED MEMORY SPACES IN AN OBJECT-ORIENTED REAL-TIME SIMULATION

David W. Geyer, Michael M. Madden*, Patricia C. Glaab,
Kevin Cunningham*, P. Sean Kenney, Richard A. Leslie

Unisys Corporation
NASA Langley Research Center
MS 169
Hampton, Virginia 23681

Abstract

Sharing memory spaces between parallel processes is a common practice in a real-time simulation environment. In an environment where parallel processes can exist on the same machine or on different machines, it is a challenge to develop and maintain reusable software that interfaces with shared memory spaces. The two main problems involved in dealing with shared memory spaces are: 1) providing platform independent access to the memory space, and 2) providing to all processes a consistent view of the structure and content of the memory space. Object-oriented techniques were used to create a software architecture designed to manage shared memory spaces. Object-oriented design patterns were used to present client code with a conceptual model of shared memory spaces while concealing the underlying implementation details. The resulting architecture is largely computing platform independent, with platform specific code being isolated to a few selected classes. The design was implemented in C^{++} for the NASA Langley Standard Real-Time Simulation (LaSRS^{++}) Application Framework. This paper provides an overview of the design and implementation of the LaSRS^{++} shared memory management software.

Introduction

This paper describes a general object-oriented software architecture designed to manage shared memory spaces in a platform independent manner. This architecture focuses on the management of specific blocks of memory within a shared memory space. The software is designed to allow client code to query for existing memory blocks or to create new memory blocks. There is no provision (currently) to delete existing memory blocks. The architecture only provides the database functionality of keeping track of existing memory blocks and creating new memory blocks. It is the responsibility of other classes in the LaSRS^{++} framework to deal with how the memory blocks are actually used by client code.

Shared Memory Spaces

A process, or task, is the executing instance of a program. Each process is composed of an addressable memory space, a set of machine instructions to execute, and other entities maintained by the operating system. It is possible to map, or overlay, a portion of this memory space with the memory space from another process. Most modern operating systems provide mechanisms to perform this mapping with another process on the same machine. This mapping is known as shared memory and is part of the more general operating system facility known as Interprocess Communication. Both processes view the shared memory space as part of their own addressable memory space. As a result, any changes to the contents of the shared memory space by one process are

*Senior Member, AIAA

visible to the other process. This type of shared memory will be referred to as physical shared memory.

A similar type of shared memory space can be achieved between processes on different machines with the use of specialized hardware. One example of this type of specialized hardware uses a networking concept known as replicated shared memory.[1] Replicated shared memory uses a serial ring network with local memory modules at each network node. Device driver software allows a given process to map the local replicated memory for the respective node into the memory space of the process. This makes the replicated memory appear just like physical shared memory. When a process writes a value to a location in the replicated memory space, the network card reflects the value to the replicated memory space of all the other nodes in the ring. This is accomplished by transmitting both the value and the memory offset of the value across the network. As a result, the value is written to the same memory offset in all of the cards on the network.

It is also possible to pass interrupts between CPUs using a replicated shared memory network. A given CPU can transmit and/or receive interrupts based on changes made to shared memory locations. The same mechanism that is used to transmit data is also used to transmit interrupts to every CPU on the ring. It is up to the software running on each and every CPU to determine the effect of the interrupt. A CPU will only receive interrupts for shared memory locations for which the CPU actually chooses to receive interrupts. A CPU can ignore or handle interrupts based on its own criteria.

The replicated shared memory approach combines the benefits of physical shared memory with a fast message passing network. The result is fast internode communication speed with little software overhead. Bohman[1] provides a more detailed discussion of the benefits of combining physical shared memory with message passing.

SCRAMNet$^+$ Network

The **S**hared **C**ommon **R**andom **A**ccess **M**emory **N**etwork (SCRAMNet$^+$), produced by Systran Corporation, is a commercial implementation of a truly general purpose replicated shared memory network. The nodes in a SCRAMNet$^+$ ring can be connected via coaxial or fiber optic cable. The delays incurred when communicating between nodes is on the order of microseconds.

Systran provides a library of low-level C callable functions for the various operating systems that might be used on a SCRAMNet$^+$ ring. The interface to these functions can be different, depending upon the platform that is being used. In order to write client code that is as platform independent as possible, it is necessary to normalize the interface to the SCRAMNet$^+$ low-level functionality.

Platform Independent SCRAMNet$^+$ Access

A SCRAMNet$^+$ network allows a group of heterogeneous processors to function together as if part of a single multiprocessing machine. Each different machine on the SCRAMNet$^+$ ring often uses its own operating system and a distinct version of the SCRAMNet$^+$ device driver. Using object-oriented techniques, it is possible to present users on all platforms with a common low-level interface to SCRAMNet$^+$ but allow the actual low-level implementation to vary according to the platform being used. This is accomplished by using several different object-oriented design patterns.[2] Design patterns describe simple and elegant solutions to specific problems in object-oriented software design.

Bridge Pattern

The Bridge pattern decouples an abstraction from its implementation. This design uses the Bridge pattern to isolate client code from the platform specific details of a specific implementation. The approach is to have clients use an abstraction object that forwards its public member function calls to a hidden platform specific implementation object. The abstraction object uses the implementation object through the pure polymorphic interface defined by the abstract implementation base class. In this case, a SCRAMNet$^+$ interface object interacts with a platform specific SCRAMNet$^+$ implementation object through a polymorphic interface. An appropriate concrete implementation class is defined for each platform being used on the SCRAMNet$^+$ ring. The appropriate concrete implementation object for a given platform is selected at run-time.

Abstract Factory Pattern

The Abstract Factory pattern is used to create the correct instance of the platform specific SCRAMNet$^+$ implementation object at run-time. This creational pattern provides an interface for creating families of related objects without specifying their concrete classes. In this case, the family of objects are the platform specific implementation objects. The abstract factory object contains knowledge of the specific platform that is being used. The constructor for the SCRAMNet$^+$ interface object invokes the `makeScramnetImpl` member function of the abstract factory object. This member function uses knowledge of the specific platform to return the appropriate implementation object for the given platform. This specific implementation object is stored as a hidden attribute of the SCRAMNet$^+$ interface object.

Singleton Pattern

The Singleton creational pattern is used whenever it is necessary to ensure a class only has one instance, and provide a global point of access to the single instance. The SCRAMNet$^+$ network used for real-time simulation at NASA Langley is designed such that any given machine contains at most one SCRAMNet$^+$ network card. For any process, there is a one-to-one correspondence between the SCRAMNet$^+$ interface class and the actual SCRAMNet$^+$ network card. Therefore, any given process on such a machine only needs a single instance of the SCRAMNet$^+$ interface class. So, the SCRAMNet$^+$ interface class is implemented as a singleton.

For simplicity, the Singleton pattern is also used for the abstract factory that creates the SCRAMNet$^+$ implementation object. This abstract factory is really just a constructor-time detail of the SCRAMNet$^+$ interface class. Using the Singleton pattern allows the SCRAMNet$^+$ interface class constructor to access the abstract factory directly.

Figure 1 shows the class diagram for the SCRAMNet$^+$ low-level interface software. Notes are included to point out the classes that utilize the various design patterns.

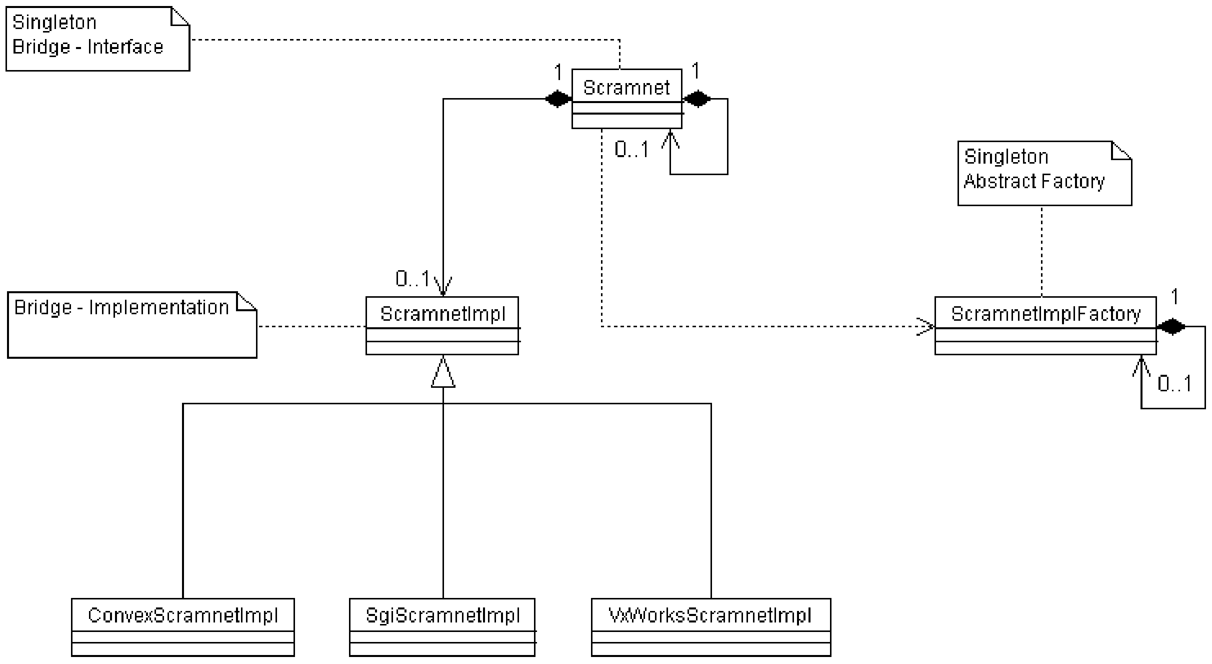

Figure 1: SCRAMNet+ Low-Level Interface Class Diagram

Design Advantages

This design has several advantages that address the issues of maintainability and portability:

1. *Client code is decoupled from the platform specific SCRAMNet$^+$ implementation details.* This decoupling allows changes in the SCRAMNet$^+$ implementation classes to have no impact on the client code; i.e., the client code does not need to be recompiled if the implementation changes. By coupling client code only to a common interface, the client code becomes platform independent. This allows client code to be portable to any platform supported by the SCRAMNet$^+$ interface class.

2. *Only a single class needs to be written to support a new platform.* Adding support for a new platform involves two steps:

   (a) Write a new SCRAMNet$^+$ implementation class that interfaces with the platform specific SCRAMNet$^+$ device driver

   (b) Add to the SCRAMNet$^+$ implementation abstract factory the capability to create the new SCRAMNet$^+$ implementation object

3. *Creation of platform specific objects can be isolated to a single abstract factory object.* The SCRAMNet$^+$ interface class only references the abstract SCRAMNet$^+$ implementation base class. All concrete implementation details are confined to the implementation abstract factory.

Managing Shared Memory

The problem of managing a shared memory space involves providing all processes a consistent view of the structure and content of the memory space. It is assumed that a memory space is being shared between processes on the same or different machines. It is also assumed that the shared memory space is going to be subdivided into multiple separate memory blocks. Therefore, the processes are going to require:

- the ability to acquire information concerning the memory blocks inside the memory space

- the ability to create new memory blocks inside the memory space

Design Strategy

The strategy used to satisfy these requirements involves using part of the shared memory space as a record keeping area to keep track of the allocated memory blocks. As the shared memory space is subdivided into smaller blocks, a record is generated for each memory block and stored in the record keeping area. Each record contains: block name, block type, block size in bytes, and block offset from starting address of the shared memory space. The record keeping area allows different processes access to information concerning the current usage of the shared memory space.

The shared memory space is divided into the following three areas:

1. System Area
   This area has a fixed size area and is used for data that pertains to all memory blocks.

2. Data Table Area
   This area is a data structure composed of bookkeeping values and a variable-length list of records that describe the arrangement of data blocks in the Data Area. Each record describes a separate, distinct memory block.
   Each record is composed of:

   - a fixed sized string for the block name
   - an integer to denote a user defined type for the memory block
   - the size of the block in bytes
   - the offset of the block from the beginning of the memory space

3. Data Area
   This area is a where the actual memory blocks are allocated.

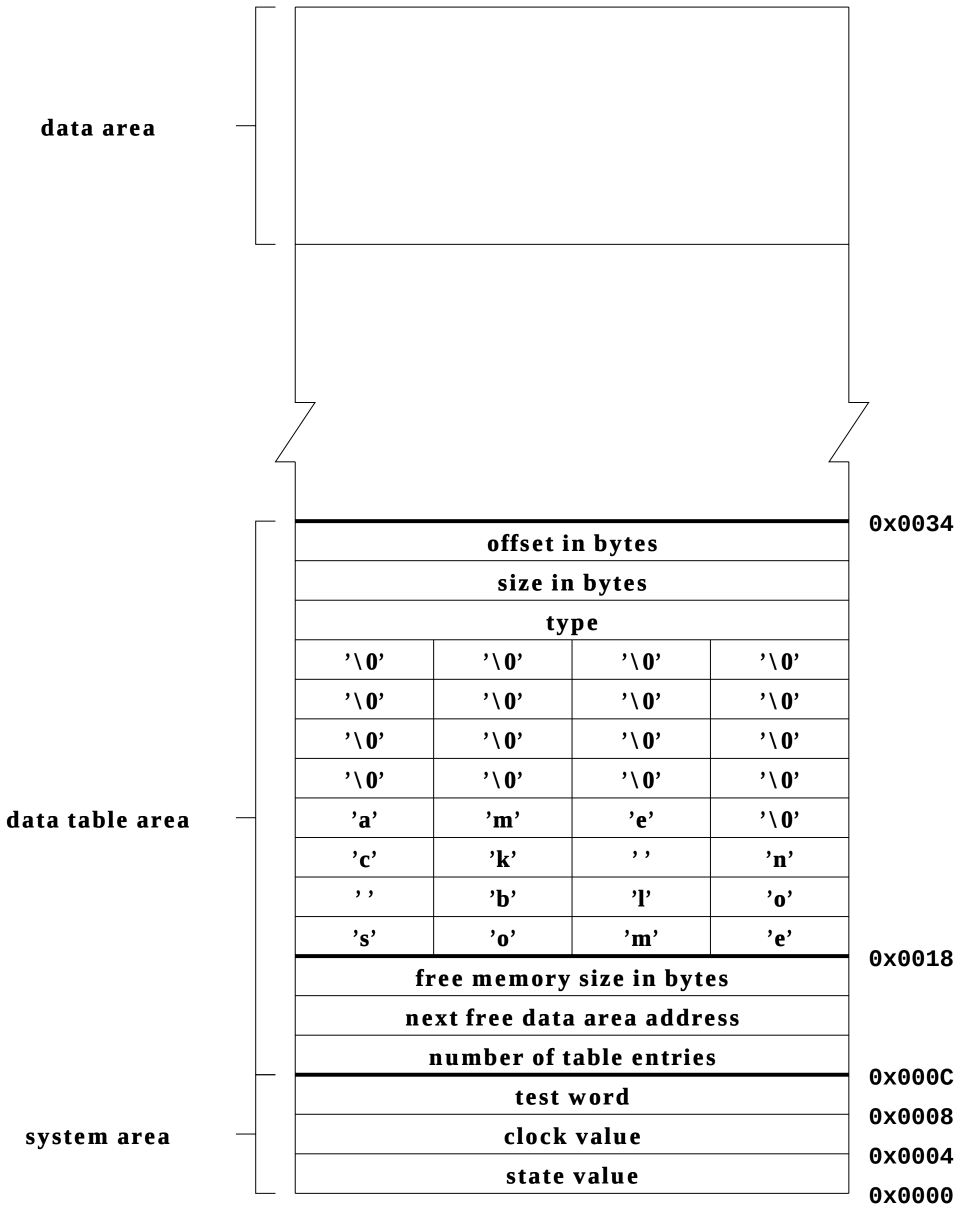

Figure 2: Three area partitioning of shared memory space

Figure 2 shows the internal structure of a sample managed memory space. The base address of the memory space is at the bottom of the diagram and the hexadecimal numbers on the right side are offsets from the base address. The system area is always located at the beginning of the shared memory space. New data records are added following the system and bookkeeping areas toward the end of the memory space. New memory blocks are sequentially allocated from the end of the memory space toward the data records. This arrangement gives the memory block management scheme maximum flexibility in allocating a variable number of variable length memory blocks.

High Level Design

The three area strategy described above is encapsulated in the `MemoryBlockManager` class. A `MemoryBlockManager` object applies the strategy to any raw memory space with a known base address and size in bytes. The `MemoryBlock` class is an abstraction of an allocated block of memory. Every memory block has an associated name, a base address and a size in bytes. The interface of the `MemoryBlockManager` class uses `MemoryBlock` objects to convey information about allocated blocks to client code.

The `MemoryBlockManager` class responds to successful memory block requests by returning one or more `MemoryBlock` objects, depending on the request criteria. The request criteria can be:

- memory block name
- all memory blocks of a specified type
- all memory blocks associated with a given machine name
- all memory blocks of a specified type associated with a given machine name
- all memory blocks in the memory space

A single `MemoryBlock` object is returned by a successful create memory block call. Client code provides the new memory block name, type and size in bytes.

Managing SCRAMNet$^+$ Memory

Managing a SCRAMNet$^+$ memory space is now simply a matter of combining the SCRAMNet$^+$ interface object with an instance of a `MemoryBlockManager` object. The `ScramnetMemoryBlock` class (derived from the `MemoryBlock` class) is also available that combines the notion of a memory block with the ability to interrupt enable/disable SCRAMNet$^+$ memory locations. This abstraction is useful in decoupling client code from the SCRAMNet$^+$ interface singleton. In practice, client code usually requests a `ScramnetMemoryBlock` object and selects memory locations inside the memory block to either send or receive interrupts. The client code does not need access to the majority of the SCRAMNet$^+$ interface. The `ScramnetMemoryBlock` class combines this functionality into a single, easy to use class.

There are situations where it is necessary to prohibit memory block creation for certain processes on a SCRAMNet$^+$ ring. This can occur for processes that are responsible only for monitoring or recording SCRAMNet$^+$ activity. It can also occur in cases where centralized control of memory block creation is needed. The creation of memory blocks may be part of the overall system initialization process. In order to facilitate this level of restricted access, two separate, but related, classes are available.

The `ScramnetReader` class provides the following functionality:

- uses the SCRAMNet$^+$ interface object to apply a `MemoryBlockManager` object to the SCRAMNet$^+$ memory space
- forwards existing memory block requests to the same `MemoryBlockManager` object
- excludes the creation of memory blocks from its own interface to ensure access only to existing memory blocks
- provides an enumerated list of types for the memory blocks in the SCRAMNet$^+$ memory space (These enumerators are used when requesting a memory block by type or when creating a memory block)

The `ScramnetReader` class uses the Singleton pattern in order to maintain a one-to-one correspondence with the SCRAMNet$^+$ interface object and hence the single SCRAMNet$^+$ network card in the machine.

The `ScramnetManager` class uses the `ScramnetReader` singleton to provide access to existing memory blocks. `ScramnetManager` also provides a create memory block request facility for client code by forwarding such requests to the `MemoryBlockManager` object used by `ScramnetReader`. The `ScramnetManager` class uses the Singleton pattern for the same reason as the `ScramnetReader` class. Both `ScramnetReader` and `ScramnetManager` return `ScramnetMemoryBlock` (i.e., specialized `MemoryBlock`) objects in response to request/create memory block calls.

Figure 3 presents a detailed class diagram for the SCRAMNet$^+$ Memory Management software.

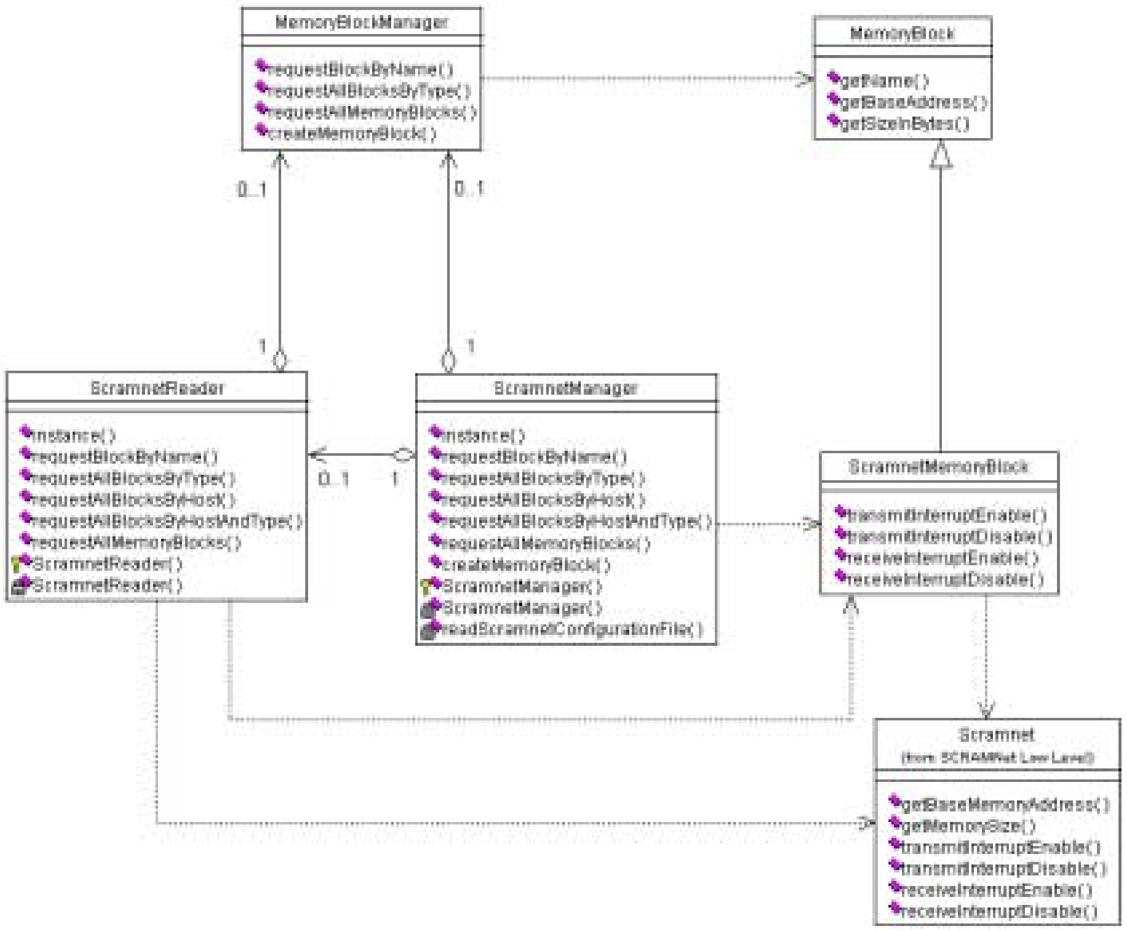

Figure 3: SCRAMNet+ Memory Management Class Diagram

Implementation Issues

Figure 4 presents a scenario diagram illustrating how the `ScramnetReader` singleton forwards an existing memory block request to the more generic `MemoryBlockManager` object. In similar fashion, Figure 5 is a scenario diagram illustrating how `ScramnerManager` singleton forwards a new memory block creation request to the same `MemoryBlockManager` object.

SCRAMNet$^+$ objects are passive, so there is no direct communication between SCRAMNet$^+$ objects executing on different processors. There is no notion of mutual exclusion between different nodes on a SCRAMNet$^+$ ring. This means that a given node can not acquire exclusive access to a SCRAMNet$^+$ memory location. There is no guarantee that a value written to a SCRAMNet$^+$ memory location will not be overwritten by some other node on the ring. This issue is not dealt with by the SCRAMNet$^+$ memory management software. Another layer of software is required that controls when the memory management objects are constructed and used. The system area at the start of the SCRAMNet$^+$ memory space can be used transmit communication protocol information between SCRAMNet$^+$ nodes.

One way to handle this issue is to have only one node in the ring be responsible for creating new SCRAMNet$^+$ memory blocks via `ScramnetManager`. All of the other nodes on the ring access existing memory blocks using `ScramnetReader`.

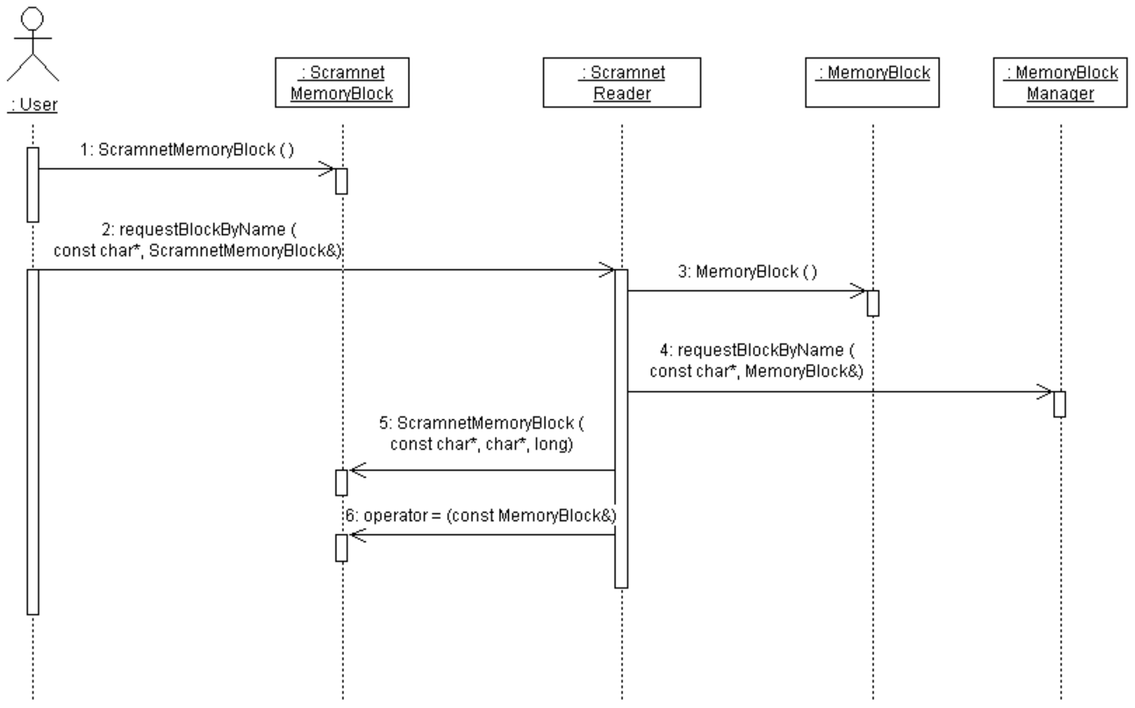
: User
: Scramnet MemoryBlock
: Scramnet Reader
: MemoryBlock
: MemoryBlock Manager
1: ScramnetMemoryBlock ( )
2: requestBlockByName (const char*, ScramnetMemoryBlock&)
3: MemoryBlock ( )
4: requestBlockByName (const char*, MemoryBlock&)
5: ScramnetMemoryBlock (const char*, char*, long)
6: operator = (const MemoryBlock&)

Figure 4: Requesting an existing SCRAMNet+ memory block by name

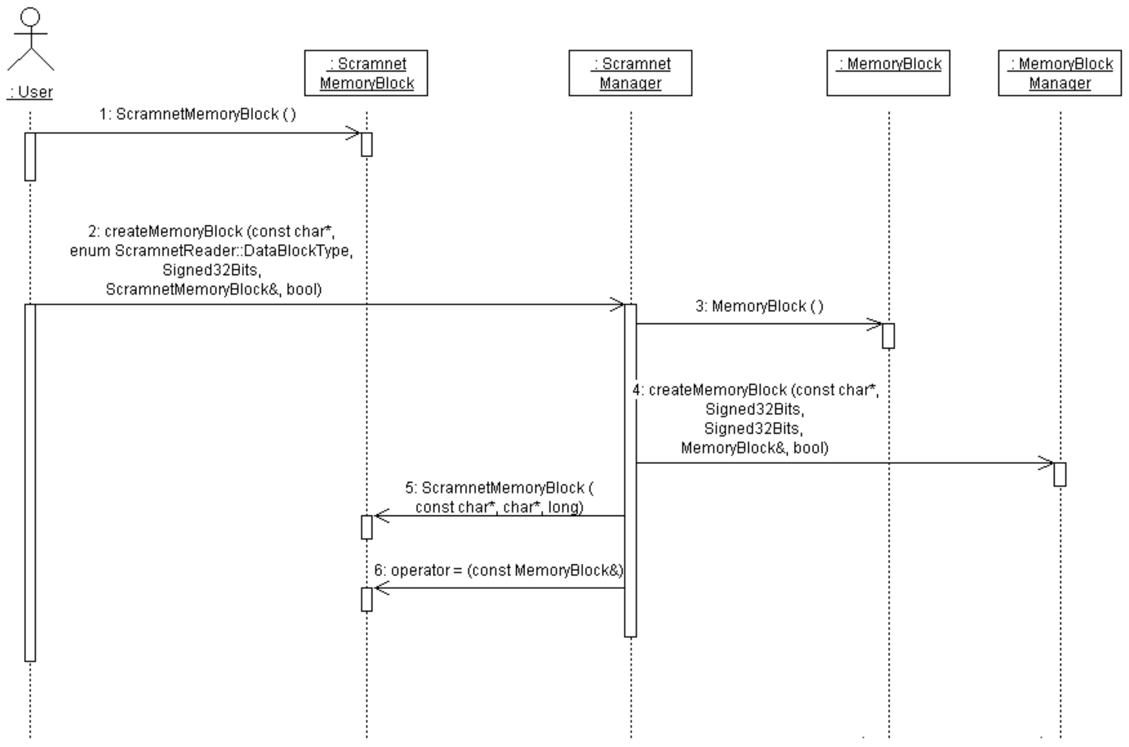
: User
: Scramnet MemoryBlock
: Scramnet Manager
: MemoryBlock
: MemoryBlock Manager
1: ScramnetMemoryBlock ( )
2: createMemoryBlock (const char*, enum ScramnetReader::DataBlockType, Signed32Bits, ScramnetMemoryBlock&, bool)
3: MemoryBlock ( )
4: createMemoryBlock (const char*, Signed32Bits, Signed32Bits, MemoryBlock&, bool)
5: ScramnetMemoryBlock (const char*, char*, long)
6: operator = (const MemoryBlock&)

Figure 5: Creating a new SCRAMNet+ memory block

Conclusions

A general-purpose method for managing shared memory spaces has been developed in an object-oriented environment. This management scheme is designed to be platform independent and has been utilized on a variety of different computing systems. The design of the software is constructed using well known object-oriented design patterns.

An object-oriented design is presented that normalizes access to SCRAMNet+ hardware across a group of heterogeneous computing platforms. Platform specific SCRAMNet+ code is isolated from the SCRAMNet+ interface code. This allows for greater reuse of client code that uses SCRAMNet+ since the client code is not directly dependent on the platform specific SCRAMNet+ code.

This general shared memory management design and the SCRAMNet+ low-level interface design are combined as the basis of a specific design to manage SCRAMNet+ memory spaces. The specific design uses a combination of composition and inheritance to achieve the desired functionality.

Bibliography

[1] T. Bohman. Shared-memory computing architectures for real-time simulation - simplicity and elegance. Technical Report D-T-SP-TPAIAA01-A-0-A1, Systran Corporation, February 1996.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[3] R. Martin. *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice-Hall, Inc., 1995. ISBN 0-13-203837-4.

[4] S. Meyers. *Effective C++*. Addison-Wesley Publishing Company, 1992. ISBN 0-201-92488-9.

[5] T Quatrani. *Visual Modeling With Rational Rose and UML*. Addison-Wesley Publishing Company, 1998. ISBN 0-201-31016-3.

[6] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, third edition, 1997. ISBN 0-201-88954-4.

[7] Systran Corporation. *SCRAMNet Network Programmer's Reference Guide*, June 1996. Document No. C-T-MR-PROGREF#-A-0-A4.